UNICEN
Universidad Nacional del Centro
de la Provincia de Buenos Aires

FACULTAD DE
CIENCIAS EXACTAS
UNICEN

Trabajo Práctico Especial

Segunda etapa

PROGRAMACIÓN III

TUDAI - Facultad de Ciencias Exactas - UNICEN

Alumnos: Gentil, Ricardo rgentil@alumnos.exa.unicen.edu.ar
Lauge, Guillermina glauge@alumnos.exa.unicen.edu.ar

Fecha de entrega: 2 de julio de 2023

Introducción

El presente trabajo, correspondiente a la asignatura Programación III -TUDAI-, aborda la implementación de un servicio que, basado en una determinada información de entrada -consistente en un listado de estaciones y distancia entre cada una de ellas con el resto-, resuelva la búsqueda de la red de menor longitud posible que logre una conexión completa entre dichas estaciones (es decir, que desde cada una de ellas pueda llegarse al resto de manera directa o indirecta), retornando la lista de túneles subterráneos a construir para tal fin. La implementación del servicio debe ser resuelta mediante dos métodos algorítmicos distintos, a saber Greedy y Backtracking, a fin de poder comparar las complejidad computacional, pros y contras de las soluciones provistas por cada uno de estos métodos.

A continuación, se ofrece una exposición detallada sobre las decisiones de implementación propuestas para la resolución de tal servicio y el análisis de las complejidades estructurales y computacionales que las fundamentan.

Desarrollo

Estructura general

Dado que el servicio solicitado requiere la implementación de una misma tarea mediante dos algoritmos distintos que consumen la misma información de entrada, se decide implementar una clase general denominada *ServicioSubterraneos* que funcione como un administrador entre la solicitud del cliente y su resolución, de modo que el usuario pueda acceder fácilmente a una solución, seleccionando simplemente el dataset y método o algoritmo a utilizar. De esta manera, la clase permite generar un encapsulamiento que diferencie la implementación del servicio de su utilización, protegiendo los detalles de la primera y simplificando la segunda.

Internamente, la clase *ServicioSubterraneos* almacena como atributos propios:

- un lector *CSVReader,* al que delega la lectura y procesamiento del dataset seleccionado por el cliente, generando un *Estado* que almacena el conjunto de estaciones a conectar, el listado de túneles disponibles para seleccionar y la cantidad de kilómetros totales de dichos túneles disponibles para ser seleccionados;
- un arreglo con la dirección de cada uno de los tres datasets ofrecidos;
- un int que representa el dataset actual, seleccionado por el cliente;
- un *Timer* que permite medir la complejidad temporal de cada algoritmo;
- un arreglo con los algoritmos disponibles, Greedy y Backtracking.

Flujo del servicio

Al recibir una solicitud por parte del cliente, el servicio constata, por un lado, que se haya seleccionado un dataset existente (de no haberlo hecho, se utiliza por defecto el dataset 1) y, por otro, si se ha seleccionado un algoritmo en particular (de lo contrario, se ejecutan ambos).

Luego, el servicio imprime el estado inicial obtenido del lector y llama a un método interno que delega al algoritmo correspondiente la búsqueda de una solución, enviando como parámetro un estado generado por el lector.[1] Este mismo método interno controla la

[1] Dado que los algoritmos manipulan y procesan la información encapsulada en el estado, se decide crear una nueva instancia de dicho estado por cada llamada al método principal de un algoritmo *hallarRedDeMenorLongitud(Estado e)*. Esto asegura que la información de entrada con que trabaja el algoritmo es exacta y simplifica la estructura general, evitando crear un nuevo estado parcial, con la misma información del estado inicial, pero que pueda ser procesada. La complejidad final derivada de leer y generar un nuevo estado es de O(t) donde t es la cantidad de túneles o entradas del dataset.

medición temporal consumida por cada método y delega la impresión de la solución a un método interno destinado a tal fin.

Procesamiento de la información de entrada

Dado que la información de entrada puede ser fácilmente asociada a la estructura de un grafo no dirigido, en el cual los vértices son las estaciones y los arcos, las distancias entre ellas, y dado que el servicio requiere el retorno de una lista de túneles solución, se decide implementar una clase llamada *Túnel* que almacena la información provista mediante tres atributos: *origen, destino, etiqueta*.

Dado que las estaciones están representadas en la información de entrada mediante una concatenación de string e int del tipo “E1”, “E2”... Se decide almacenar las estaciones origen y destino en la clase Túnel simplemente como el Integer que las representa, a fin de evitar crear una clase para tal fin, que sólo almacenaría el nombre provisto en la información de entrada y, por lo tanto, en este contexto, sería redundante e innecesaria, complejizando innecesariamente la resolución del servicio.

Asimismo, dado que las distancias vienen representadas como valores enteros positivos, se elige el tipo Integer, aunque, en otro contexto, sería más acertado utilizar un tipo de dato más exacto (double, float).

Estado

Se utiliza una clase llamada *Estado* para delegar la creación y almacenamiento de la información de entrada y ser transferida a los algoritmos para procesarla y retornar una solución. Para ello, es inicializado por el lector con tres atributos propios:

- HashSet de estaciones de estaciones a conectar. Se utiliza esta estructura dada la facilidad de inserción de elementos sin necesidad repeticiones. Dado que en esta estructura sólo se realizan operación de inserción y nunca se recorre, la complejidad relacionada a ella es constante.

- LinkedList de túneles disponibles. Esta estructura se utiliza para insertar y eliminar elementos en operaciones cuya complejidad es constante, sólo son recorridas en caso de ser impresas, donde la complejidad es O(n).

- kilómetros disponibles. Variable que lleva la cuenta del kilometraje total de la lista de túneles disponibles.

Al ser recibido por un algoritmo, éste inicializa tres atributos más destinados al procesamiento de la información y almacenaje de la solución:

- LinkedList de túneles seleccionados
- kilómetros seleccionados
- *UnionFind*. Clase específica para el tratamiento de operaciones de conjuntos utilizada por ambos algoritmos.

Estructura Union Find

Este tipo de estructura, también llamada Disjoint-set Union es utilizada para manejar las estaciones a conectar como conjuntos.

Al ser instanciada por un algoritmo, inicializa dos arreglos de tamaño equivalente a las estaciones a conectar, en los que se almacenan, por un lado, los padres o antecesores (en principio, cada estación refiere a sí misma), por otro, los rangos de cada conjunto (inicializados en 0), equivalente a la altura del árbol que representan si se los ve como tales. Dado que al conectarse dos conjuntos uno pasa a ser parte del otro, se utiliza esta última variable para optimizar la unión determinando en qué conjunto se integran los elementos para que las cargas queden balanceadas.

Esta clase cuenta con los siguientes métodos:

- *conectarEstaciones(tunel.getOrigen(), tunel.getDestino()),* representa la conexión entre dos estaciones -al seleccionar un túnel determinado-;

- *estanConectadas(tunel_actual.getOrigen(), tunel_actual.getDestino()),* verifica si dos estaciones pertenecen a un mismo conjunto;

- *conexionCompleta(),* verifica si todas las estaciones son parte de un mismo set o conjunto.

La complejidad computacional de estos métodos es O(e), donde e es la cantidad de estaciones.

Algoritmos

Se decide implementar una clase abstracta *Algoritmo* de la que extienden las clases *Greedy* y *Backtracking* a fin de generalizar y simplificar la funcionalidad del servicio, evitando la duplicación de código. Los algoritmos son instanciados junto con el servicio y almacenados por él a fin de evitar la creación nuevas instancias en cada llamado al método que los utiliza.

Greedy

Criterio de selección

En primer lugar, el algoritmo propuesto ordena de manera ascendente el arreglo de túneles disponibles recibido como parámetro de entrada en base a las distancias cubiertas por cada uno de ellos. Para ello, se implementa la interfaz *Comparable* y el método *compareTo* en la clase Túnel.

Posteriormente, en cada iteración, selecciona de manera ordenada los túneles disponibles -retornando el elemento eliminado, siempre el primero de esta lista, y actualizando el kilometraje de la misma-.

Análisis de factibilidad

Si se verifica que el túnel seleccionado conecta estaciones que aún no se encuentran conectadas, se selecciona el túnel, es decir, se lo añade a la lista de túneles seleccionados que será retornada como solución, y se actualiza el kilometraje de la misma.

Condición de corte de la iteración: El algoritmo repetirá las operaciones de selección y análisis de factibilidad del elemento seleccionado tantas veces como sea necesario, hasta que se haya logrado una conexión completa entre las estaciones o no haya más túneles disponibles.

Estructuras de almacenamiento y complejidad computacional

Se elige como estructura de almacenamiento de los túneles disponibles y seleccionados una LinkedList, implementada en Java como una lista doblemente vinculada, que permite realizar la eliminación e inserción de elementos al principio y el final de la lista con una complejidad de O(1), es decir, constante, mientras que la operación de mostrarlos será siempre O(n) -donde n es la cantidad de túneles solución- sea utilizada esta estructura o un ArrayList. Se descarta el uso de esta última estructura dado que, al funcionar internamente como un array, supondría un aumento innecesario de la complejidad computacional.

Por otra parte, se utiliza sólo una vez el método *sort* de la interfaz *Collections* de Java que, a partir de Java 8, incluye la variante *Timsort*, una implementación que utiliza por defecto una variante del algoritmo *mergesort,* cuya complejidad es O(n log n) y, aunque no mejora la complejidad temporal, optimiza el uso de memoria.

Se añade a las complejidades mencionadas aquellas derivadas de las operaciones realizadas por la clase *UnionFind:* en cada iteración, verificar si las estaciones origen y

destino del túnel seleccionado ya se encuentran conectadas, y constatar si hay una conexión completa. Estas operaciones tienen una complejidad casi lineal O(e), donde e es el número de estaciones.

En suma, la complejidad computacional del algoritmo greedy propuesto es lineal, O(t), donde *t* es la cantidad de túneles disponibles,[2] dependientes de la información de entrada de cada dataset (ver detalle en tabla comparativa en anexo), ya que en el peor de los casos se considerarán todos los túneles disponibles.

Backtracking

Condición de corte y selección de estado solución

```
if (conexionCompleta) {

    if (estado_actual.getKmSeleccionados() < this.solucion.getKmSeleccionados()) {
        this.solucion.setTunelesSeleccionados(new LinkedList<>(estado_actual.getTunelesSeleccionados()));
        this.solucion.setKmSeleccionados(estado_actual.getKmSeleccionados());
    }

} else {
```

Este algoritmo utiliza la misma condición de corte utilizada por greedy, verificación de conexión completa y existencia de túneles disponibles. Una vez alcanzado un estado final, se considera solución si la distancia cubierta por los túneles es menor a la ofrecida por la mejor solución almacenada, que es inicializada con la suma de las distancias de todos los túneles disponibles.

Aunque aquí no se ha implementado por no considerarse necesario, cabe notar que cuando se trata de datos de entrada de gran tamaño, podría utilizarse el resultado arrojado por el algoritmo greedy para optimizar el funcionamiento del backtracking, seteando este valor como primera solución y utilizándolo como estrategia de poda para limitar las conmutaciones de túneles posibles. En este caso, el ordenamiento previo de la lista de túneles de entrada pierde sentido, dado que igualmente se analizarán todas las permutaciones posibles.

Decisiones que conforman el árbol de búsqueda

Por cada túnel disponible, se consideran dos opciones posibles: agregar o no agregar el túnel a la lista de túneles seleccionados. De esta manera, se genera un árbol de búsqueda binario.

[2] Ordenamiento O(t log t) + [Iteraciones O(t) * (Selección O(1) + Conectadas O(e) + Conexión completa O(e))] donde t = túneles disponibles, e = estaciones a conectar.

Estrategias de poda

El algoritmo propuesto utiliza tres técnicas optimización como condiciones de recursión en el caso de agregar el túnel a la selección:

```
if ((estado_actual.getCantidadTunelesSeleccionados() + 1) < estado_actual.getCantidadEstacionesAConectar()) {
    if (estado_actual.getKmSeleccionados() + tunel_actual.getEtiqueta() < this.solucion.getKmSeleccionados()) {
        if (!estado_actual.estanConectadas(tunel_actual.getOrigen(), tunel_actual.getDestino())) {
```

a) que la cantidad de túneles seleccionados sea menor a la cantidad de estaciones a conectar;

b) que la distancia total de los túneles seleccionados sea menor a la distancia de la solución almacenada;

c) que las estaciones origen y destino del túnel no se encuentren conectadas;

Tanto en el caso de agregar como en el de no agregar el túnel a la selección actual, se utiliza una estrategia de poda que permite reducir notablemente el número de recursiones del algoritmo, evitando recursar cuando no hay más túneles disponibles.

```
conexionCompleta = estado_actual.conexionCompleta();
if (estado_actual.hayTunelesDisponibles() || conexionCompleta ) {
    _back(estado_actual);
}
```

Observación sobre el tratamiento de Union Find

Dados los beneficios de utilizar esta estructura, pero la complejidad de implementar un método que permita deshacer una unión una vez que el algoritmo regresa de una recursión, se decide almacenar el objeto *UnionFind* con que cuenta el estado antes de iniciar la recursión y reemplazar sus valores -arreglo de padres y rangos- al volver. Esto supone un aumento de la complejidad total de O(e), donde *e* es el número de estaciones, por cada recursión -$O(2^t)$, donde *t* es la cantidad de túneles disponibles-.

No obstante, dado que la cantidad de estaciones siempre es menor a la cantidad de túneles, puede decirse que la complejidad total del método es exponencial, de $O(2^t)$.[3]

[3] [Iteraciones $O(2^t)$ * (GetTunel O(1) + Conectadas O(e) + Selección O(1) + Conexión completa O(e) + Reemplazo de Union Find O(2e))] = $O(2^t)$ * O(e) = $O(2^t)$, donde t = túneles disponibles, e = estaciones a conectar.

Conclusión

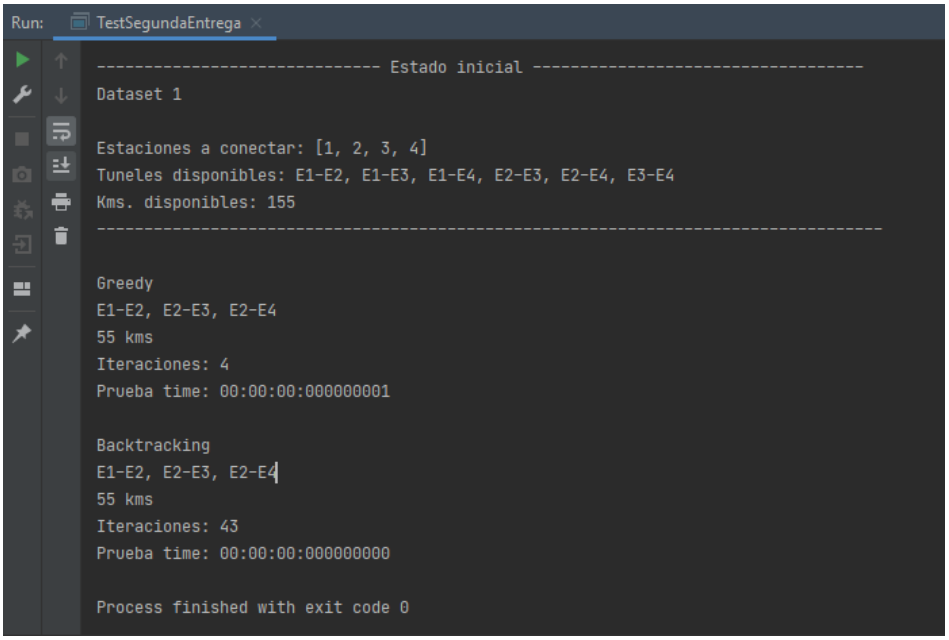
El análisis del proceso de desarrollo y resultados ofrecidos por el presente trabajo permite concluir que, en lo que respecta a los algoritmos Greedy y Backtracking, si bien en términos generales podría decirse que este último es el más adecuado porque garantiza obtener la mejor solución posible al contemplar todas las soluciones existentes, mientras que Greedy alcanza sólo una de esas posibles soluciones, cuya calidad depende en gran medida del criterio de selección utilizado, en el marco concreto del caso propuesto, Greedy no sólo logra alcanzar las mismas soluciones arrojadas por el Backtracking sino que también logra hacerlo con una complejidad computacional notablemente inferior, lineal en lugar de exponencial.

En lo que respecta a las decisiones sobre las estructuras y el diseño de la implementación, la utilización de las herramientas provistas por java siempre resulta beneficiosa dada su constante optimización y fiabilidad. La utilización de la estructura Union Find resulta sumamente beneficiosa en este caso en que deben manipularse conjuntos, evitando utilizar estructuras más complejas, como grafos, para alcanzar la solución.

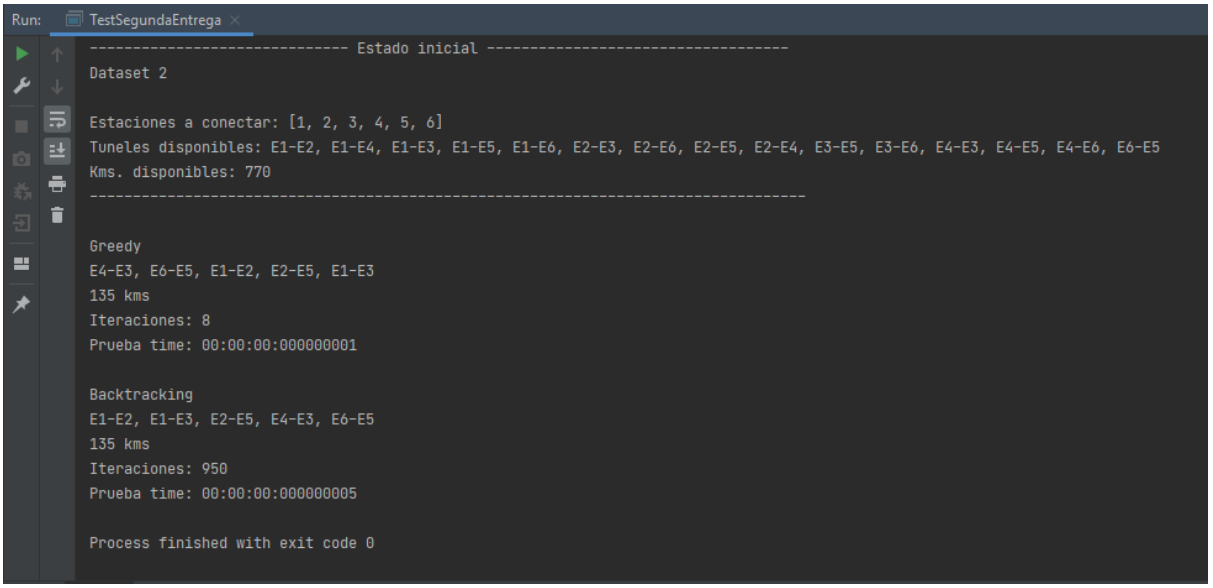
Anexo

ENTRADA			ITERACIONES	
Dataset	Estaciones	Túneles	Greedy	Backtracking
Dataset 1	4	6	4	43
Dataset 2	6	15	8	950
Dataset 3	13	78	25	3115156676

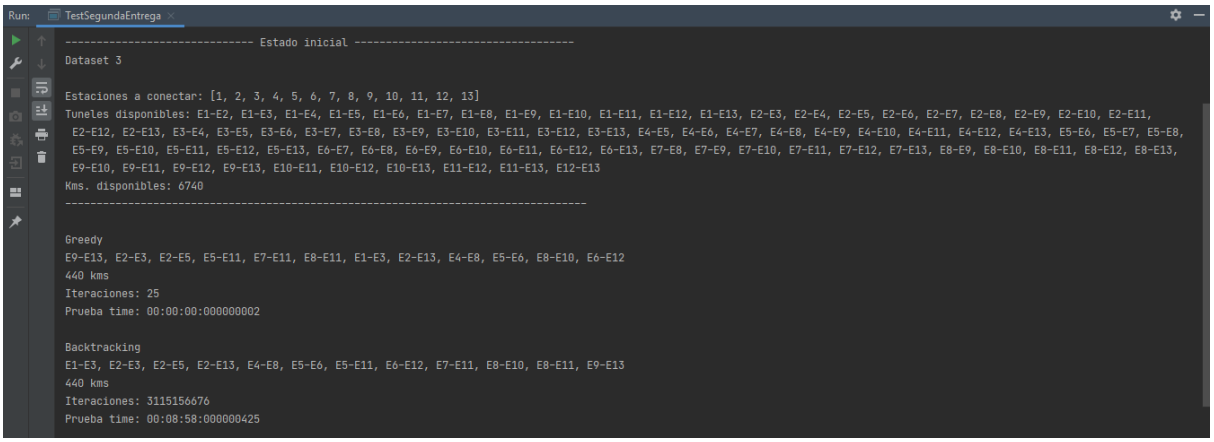
Tabla comparativa de iteraciones realizadas por cada algoritmo, por cada dataset



Dataset 1



Dataset 2


Run:
TestSegundaEntrega
---------- Estado inicial ----------
Dataset 3
Estaciones a conectar: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
Tuneles disponibles: E1-E2, E1-E3, E1-E4, E1-E5, E1-E6, E1-E7, E1-E8, E1-E9, E1-E10, E1-E11, E1-E12, E1-E13, E2-E3, E2-E4, E2-E5, E2-E6, E2-E7, E2-E8, E2-E9, E2-E10, E2-E11,
E2-E12, E2-E13, E3-E4, E3-E5, E3-E6, E3-E7, E3-E8, E3-E9, E3-E10, E3-E11, E3-E12, E3-E13, E4-E5, E4-E6, E4-E7, E4-E8, E4-E9, E4-E10, E4-E11, E4-E12, E4-E13, E5-E6, E5-E7, E5-E8,
E5-E9, E5-E10, E5-E11, E5-E12, E5-E13, E6-E7, E6-E8, E6-E9, E6-E10, E6-E11, E6-E12, E6-E13, E7-E8, E7-E9, E7-E10, E7-E11, E7-E12, E7-E13, E8-E9, E8-E10, E8-E11, E8-E12, E8-E13,
E9-E10, E9-E11, E9-E12, E9-E13, E10-E11, E10-E12, E10-E13, E11-E12, E11-E13, E12-E13
Kms. disponibles: 6740
Greedy
E9-E13, E2-E3, E2-E5, E5-E11, E7-E11, E8-E11, E1-E3, E2-E13, E4-E8, E5-E6, E8-E10, E6-E12
440 kms
Iteraciones: 25
Prueba time: 00:00:00:000000002
Backtracking
E1-E3, E2-E3, E2-E5, E2-E13, E4-E8, E5-E6, E5-E11, E6-E12, E7-E11, E8-E10, E8-E11, E9-E13
440 kms
Iteraciones: 3115156676
Prueba time: 00:08:58:000000425

Dataset 3